Poster Paper

SynthEdit: Format transformations by example using edit operations

Alex Bogatu, Alvaro A.A. Fernandes, Norman W. Paton, Nikolaos Konstantinou
School of Computer Science, University of Manchester
alex.bogatu@manchester.ac.uk

ABSTRACT

Format transformation is one of the most labor intensive tasks of a data wrangling process. Recent advances in programming by example proposed synthesis algorithms that showed promising results on spreadsheet data. However, when employed on repositories consisting of multiple sources and large number of examples, such algorithms manifest scalability issues. This paper introduces a new transformation synthesis technique based on edit operations that enables efficient learning of transformation programs. Empirical results show comparable effectiveness and dramatic improvements in efficiency over the state-of-the art.

1 INTRODUCTION

Format transformation is a sub-task of data wrangling ([3], [8]) that carries out changes to the representation of textual information, with a view to reducing inconsistencies. Such tasks are typically coded manually by experienced users through scripts that change the representation of data. Recent advances in Programming By Example (PBE) led to algorithms such as *FlashFill* [4] and *BlinkFill* [11] that synthesize transformation programs from user given input-output examples. These algorithms represent important steps towards automatic format transformation in the fields for which they have been devised: spreadsheets. Spreadsheet processing often involves datasets of manageable size, a small number of examples and active user involvement in providing additional example data when needed. In contrast, in areas of data wrangling and data analysis, the task of format transformation is applied repetitively on large datasets from many sources, where more examples are available and user supervision is impractical [2]. Increased volume of example data reveals the opportunity for taking the human factor out of the synthesis process. Automating format transformation by increasing the number of examples represents a challenge for current synthesis algorithms due to their high complexity: exponential in the number of examples and highly polynomial in the length of the examples [10].

In this paper we contribute a solution, which we call *SynthEdit*, to the problem of automatic format transformation. We are motivated by the high computational cost of current state-of-the-art techniques when the number of available examples increases, and propose an approach based on finite state automata that scales better than algorithms such as *FlashFill* when there are more than a few examples available - typically tens or hundreds. The improved efficiency comes at the expense of expressiveness, but allows *SynthEdit* to benefit from substantial example data when available, and, therefore, to cover many inconsistency cases and to eliminate the need for user involvement.

Related work: The problem of learning transformations from examples has been addressed in a number of settings ranging from restructuring of textual information [4], [11], to table formatting [6], and text extraction [1]. More complex systems such as TDE [5] embody a wider range of transformations using external information in addition to user-provided examples.

The closest to our work are algorithms such as FlashFill which have specifically been designed for spreadsheets, assume high user involvement and use DAG-based synthesis solutions that aim to cover the entire space of possible transformations between two given strings. The user's main task in such systems is to provide additional examples for cases not covered by the previous instances. The resulting algorithms have exponential complexity w.r.t. the number of examples [10].

While recent works [2], [7] have proposed techniques that minimize the user involvement in generating examples, *SynthEdit* aims to fully automate the task of format transformation for larger repositories such as data lakes, richer in potentially useful example data, and challenging to address through manual authoring of scripts due to their size and diversity.

2 PRELIMINARIES

We start by describing a representative example where our algorithm can be employed, inspired by web real world data.

Example 2.1. 1900s NY state governor names and term years:

Source	Target
Hugh Leo Carey (74-82)	Hugh L. Carey (1974-1982)
Jay Henry Lehman (33-42)	Jay H. Lehman (1933-1942)

In Example 2.1, the task is to derive a transformation that can generate the *Target* values using the *Source* values. In this paper we refer to such pairs of strings as *example instances*.

Tokens: We view a string s as a collection of *tokens*, where each token represents a sub-string of s. Similarly to existing algorithms for format transformation, e.g., *FlashFill*, *SynthEdit* supports three types of tokens: (i) regular expression tokens - that match a predefined regular expression pattern; (ii) constant string tokens - with a value equal to the corresponding constant string, and (iii) special tokens - beginning/end of a string.

Regular expression primitives: Obtaining the set of tokens for a string s builds on a set of primitive lexical classes defined by regular expressions. The regular expression primitives we use are notated as follows: N = [0-9]+, U = [A-Z]+, L = [a-z]+, A = [A-Za-z]+, Q = [A-Za-z0-9]+, P = [.,;: /-_?!&$]+, W = \s+.

Transformations: Going back to Example 2.1, we can express the source string in the first row as a collection of token types, henceforth called a *token-type representation*: A W A W A W P N P N P. Such representations will be used to derive a transformation that will produce the target string from the source string. One simple such transformation would have the following English language specification: *Replace "Leo", "74", and "82" from the source with "L", "1974", and "1982" from the target, resp.*. Although such

$$
\begin{aligned}
\textit{Regex primitive } \mathtt{r} &:= \mathtt{N} \mid \mathtt{U} \mid \mathtt{L} \mid \mathtt{A} \mid \mathtt{Q} \mid \mathtt{P} \mid \mathtt{W} \\
\textit{Position expression } \mathcal{P} &:= Pos(\mathtt{r}_1, \mathtt{r}_2, c) \\
\textit{Token } \mathtt{t} &:= (\mathtt{r}, \mathcal{P}) \\
\textit{String expression } \mathcal{E} &:= Copy(\mathtt{t}) \mid Const(\mathtt{s}) \\
&\quad \mid Substr(\mathtt{t}, i, j) \mid Concat(\mathcal{E}_1, \ldots, \mathcal{E}_n) \\
\textit{Edit operation } O &:= INS(\mathcal{E}) \mid DEL(\mathtt{t}) \mid SUB(\mathtt{t}, \mathcal{E}) \\
\textit{Transformation } \mathcal{T} &:= O_1;\ O_2;\ \ldots;\ O_n;
\end{aligned}
$$

Figure 1: Transformation language syntax

a transformation is consistent with the first example instance, it is too specific to correctly transform the string in the second row. Fortunately, we can generalize the transformation by using token types instead of actual sub-string values: *Replace the second* A*-type token, the first* N*-type token, and the second* N*-type token from the source with the first* U*-type token, the first* N*-type token, and the second* N*-type token from the target, resp.*. Now, the same transformation is consistent with both example instances.

3 TRANSFORMATION LANGUAGE

In this section we propose a transformation language that is expressive enough to describe the previous transformation using simple edit operations: *Insert* (referred to as *INS*), *Delete* (referred to as *DEL*), and *Substitute* (referred to as *SUB*). The complete syntax of the transformation language is in Figure 1, while the semantics are described below. We use the notation ϵ to denote an empty string, $len(s)$ for the length of a string s, $s[i:j]$ for the sub-string that starts at index i in s and ends at index j, and c to denote the c^{th} occurrence of a token type in a string[1].

The semantics of a position expression, $Pos(\mathtt{r}_1, \mathtt{r}_2, c)$, is to evaluate to a sub-string $s[j:k]$ of a given string s, such that $\exists\, i, j, k, l\ 0 \leq i < j < k < l \leq len(s) - 1$, where $s[i:j]$ matches $\mathtt{r}_1$ and $s[k:l]$ matches $\mathtt{r}_2$. Furthermore, $s[j:k]$ is the c^{th} such sub-string in s. If such a sub-string does not exist then the expression evaluates to ϵ. Such an expression allows us to uniquely identify each token in a given string using its neighbour tokens and, at the same time, ensures that the expression will (likely) evaluate to ϵ when applied on strings with different format representations. We can now redefine a token t as a pair consisting of a token type r and a position expression $\mathcal{P}$. Note that t only exists if the string value returned by $\mathcal{P}$ matches r.

The $Copy(\mathtt{t})$ expression evaluates to the string value of t. $Const(\mathtt{s})$ evaluates to a constant string s. $Substr(\mathtt{t}, i, j)$ returns the sub-string that starts at position i and ends at position $j-1$ of the string value of t. $Concat(\mathcal{E}_1, \ldots, \mathcal{E}_n)$ performs string concatenation on the results of the underlying string expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$. The edit operations $INS(\mathcal{E})$, $DEL(\mathtt{t})$, and $SUB(\mathtt{t}, \mathcal{E})$ perform insertion, removal, and replacement, resp. of some string resulting from the expressions given as parameters, on a given string s. Lastly, $\mathcal{T}$ is a sequence of edit operations applied on s.

Figure 2 shows 5 operations (from a total of 12) of a transformation that can be used to edit both source strings from Example 2.1 into their corresponding target strings. Considering the first row from Example 2.1, the intuition in (1) and (2) is to replace the first occurrences of an A-type token, i.e., *Hugh*, and the first white space with a copy of themselves. Similarly, in (3), the token *Leo* from the source string is replaced with the first letter of itself. In (4), a dot, ".", is inserted after the result of operation (3).

[1] All string positions, i.e. indexes, start from 0.

$$
\begin{aligned}
&SUB((\mathtt{A}, Pos(\hat{}, \mathtt{W}, 0)), Copy((\mathtt{A}, Pos(\hat{}, \mathtt{W}, 0)))); && (1)\\
&SUB((\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 0)), Copy((\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 0)))); && (2)\\
&SUB((\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 0)), Substr((\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 0)), 0, 1)); && (3)\\
&INS(Const(".")); && (4)\\
&SUB((\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 0)), Concat(Const("19"), Copy((\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 0))))); && (5)
\end{aligned}
$$

Figure 2: Transformation for Example 2.1

Operation (5) replaces the first number between two punctuation tokens, i.e., *74*, with the result of the concatenation of a constant string, *19*, and the same number to obtain *1974*. Note that the transformation in Figure 2 is applied on a copy of the source, as opposed to modifying the string in-place. This ensures that source tokens used by later operations are not lost if an early operation deletes/substitutes them.

4 SYNTHESIS ALGORITHM

In this section we describe an algorithm that, starting from the example instances provided in Example 2.1, learns the transformation from Figure 2. Given an example instance, the algorithm consists of 4 phases: 1) tokenize the source and target strings; 2) synthesize edit operations; 3) synthesize string expressions; 4) merge the results of 2) and 3) to create a transformation.

Tokenization: Both *source* and *target* strings are split into tokens using a function Tokenize. Specifically, the function searches for sub-strings that match one of the regular expression primitives defined in Section 2. Each such match represents a new token t_i. Next, by looking at the neighbouring matches, the function learns a position expression which uniquely identifies t_i in the parent string. Tokenizing the source string of the first row in Example 2.1 returns: $(\mathtt{A}, Pos(\hat{}, \mathtt{W}, \mathtt{0}))$, $(\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 0))$, $(\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 0))$, $(\mathtt{W}, Pos(\mathtt{A}, \mathtt{A}, 1))$, $(\mathtt{A}, Pos(\mathtt{W}, \mathtt{W}, 1))$, $(\mathtt{W}, Pos(\mathtt{A}, \mathtt{P}, 0))$, $(\mathtt{P}, Pos(\mathtt{W}, \mathtt{N}, 0))$, $(\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 0))$, $(\mathtt{P}, Pos(\mathtt{N}, \mathtt{N}, 0))$, $(\mathtt{N}, Pos(\mathtt{P}, \mathtt{P}, 1))$, $(\mathtt{P}, Pos(\mathtt{N}, \$, 0))$

Edit operation synthesis: The top-level transformation is a collection of edit operations parameterized with tokens and/or string expressions. Given an example instance, with its corresponding token-type representation of source (T_s) and target (T_t) derived from the token sets obtained at the previous step, a function $\mathtt{EditSynthesis}(T_s, T_t)$ generates a sequence of edit operations that edits T_s into T_t. This can be done using an edit distance algorithm, such as the one proposed in [9], where the composition operation on weighted finite state automata (WFSA) is used to generate all possible edit operations that transform the source into the target. By assigning equal weights to each edit operation, the shortest path of the lattice of operations returned by the composition of WFSA denotes the simplest way to obtain the token-type representation of the target. Note that each path of the obtained lattice denotes a valid sequence of edit operations which would produce the target string, but we are only interested in the simplest one. As an example, consider again the first row from Example 2.1 with the token-type representations T_s = A W A W A W P N P N P and T_t = A W U P W A W P N P N P. $\mathtt{EditSynthesis}(T_s, T_t)$ can produce the following sequence of edit operations[2] that transform T_s into T_t:

$$
\begin{aligned}
&SUB(\mathtt{A}^s_0, \mathtt{A}^t_0);\ SUB(\mathtt{W}^s_0, \mathtt{W}^t_0);\ SUB(\mathtt{A}^s_1, \mathtt{U}^t_0);\ INS(\mathtt{P}^t_0);\\
&SUB(\mathtt{W}^s_1, \mathtt{W}^t_1);\ SUB(\mathtt{A}^s_2, \mathtt{A}^t_1);\ SUB(\mathtt{W}^s_2, \mathtt{W}^t_2);\ SUB(\mathtt{P}^s_0, \mathtt{P}^t_1); \qquad (6)\\
&SUB(\mathtt{N}^s_0, \mathtt{N}^t_0);\ SUB(\mathtt{P}^s_1, \mathtt{P}^t_2);\ SUB(\mathtt{N}^s_1, \mathtt{N}^t_1);\ SUB(\mathtt{P}^s_2, \mathtt{P}^t_3);
\end{aligned}
$$

$\mathtt{A}^s_i/\mathtt{A}^t_j$: the i^{th}/j^{th} token of type A from source/target, $i, j \geq 0$.

Note that Expression (6) denotes the transformation between a given source and a given target, but it cannot be used directly

[2] Expression (6) omits the Pos constructors for clarity.

Algorithm 1 String expression synthesis

Input: Index entry: $t_i \rightarrow pairs_{t_i}$
Output: A string expression $\mathcal{E}$

1: **function** ExpressionSynthesis
2: **if** $pairs_{t_i} == []$ **then return** $Const(t_i)$ **end if**
3: $(s_j, lcs_j) \leftarrow$ best_pair(pairs_{t_i})
4: **if** $s_j == lcs_j == t_i$ **then** $\mathcal{E} \leftarrow Copy(s_j)$
5: **else if** $lcs_j \subset s_j$ && $lcs_j == t_i$ **then**
6: $\mathcal{E} \leftarrow Substr(s_j, \text{indexOf}(lcs_j, s_j))$
7: **else**
8: $\mathcal{E} \leftarrow Concat(\text{ConcatSynthesis}(t_i, pairs_{t_i}))$
9: **end if**
10: **return** $\mathcal{E}$
11: **end function**

Algorithm 2 *Concat* expression synthesis

Input: Index entry: $t_i \rightarrow pairs_{t_i}$
Output: A list of string expression exp

1: **function** ConcatSynthesis
2: $exp \leftarrow [\ ]$
3: **for all** $(s_j, lcs_j) \in pairs_{t_i}$ **do**
4: **if** $s_j == lcs_j$ **then** $exp \leftarrow exp + [Copy(s_j)]$
5: **else** $exp \leftarrow exp + [Substr(s_j, \text{indexOf}(lcs_j, s_j))]$
6: **end if**
7: $t_i \leftarrow \text{replace}(t_i, lcs_j, \text{""})$
8: **end for**
9: **if** $\text{len}(t_i) > 0$ **then** $exp \leftarrow exp + [Const(t_i)]$ **end if**
10: **return** exp
11: **end function**

when the target is not given. As such, our next objective is to express each target token in Expression (6) as a string expression applied on some source token, e.g., Figure 2. This will allow us to apply the newly obtained operations on new input strings.

String expression synthesis: Expressing a target token t_i as a processing of some source tokens requires the identification of the source token (or the group of source tokens) whose value(s) are the closest to t_i. To this end, we create an inverted index $\mathcal{I}$ in which each target token value t_i identifies a list of pairs of the form (s_j, lcs_j), where lcs_j is the *longest common sub-string* between a source token value s_j and t_i. The first two columns from Table 1 depict three index entries obtained for the first row of Example 2.1. The third column indicates the type of string expressions that can be applied on the source tokens to obtain t_i, as described next.

For each entry of I, we can apply a function `StringSynthesis` (defined in Algorithm 1) to synthesize a string expression that uses some source tokens to obtain the target token. The function at line 3 in Algorithm 1 returns a pair (s_j, lcs_j) where s_j is the source token value that best covers the target token value. Note that we only process the best such pair because its source token has the most useful value to derive the target token. For when the best pair is not the desired one, i.e. the target token has to be obtained from a different source token, we rely on multiple example instances to disambiguate and to generalize a transformation. The $\text{indexOf}(lcs_j, s_j)$ function at line 6 returns the start and end indexes of lcs_j in s_j.

When there is no source token that can be used to obtain t_i, `ExpressionSynthesis` returns a *Const* expression. Alternatively, when the longest common sub-string, the source and target token values are all identical, the result is a *Copy* expression, e.g., Row 1 in Table 1. If the longest common sub-string is only equal to the target token value, it means that t_i can be obtained from the source token using a *Substr* expression, e.g., Row 2 in Table 1.

Finally, a *Concat* expression is synthesized, using Algorithm 2, when the source token value is a sub-string of the target token value. The `ConcatSynthesis` function processes all pairs of the current index entry, as opposed to only the best pair, and consumes the target token value as soon as it is able to derive a sub-string of it. For example, for Row 3 in Table 1, *74* is a source token and, therefore, the condition at line 4 in Algorithm 2 is met. At the next iteration t_i = "19" (because we consume the previous value) but there are no pairs in $pairs_{t_i}$ that cover the new value which means that the condition at line 9 evaluates to true and the next expression learned is *Const*.

Transformation synthesis: The last step of the algorithm replaces the target tokens in Expression (6) with the corresponding

Table 1: Index entries

	t_i	$[(s_j, lcs_{(s_j,t_i)})]$	Expression
1	Hugh	[(Hugh, Hugh)]	*Copy*
2	L	[(Leo, L)]	*Substr*
3	1974	[(74, 74)]	*Concat*

string expressions learned by Algorithm 1. The result, listed below, is a transformation consistent with the example instance and applicable on new input strings, similar in format representation with the source string of the example instance.

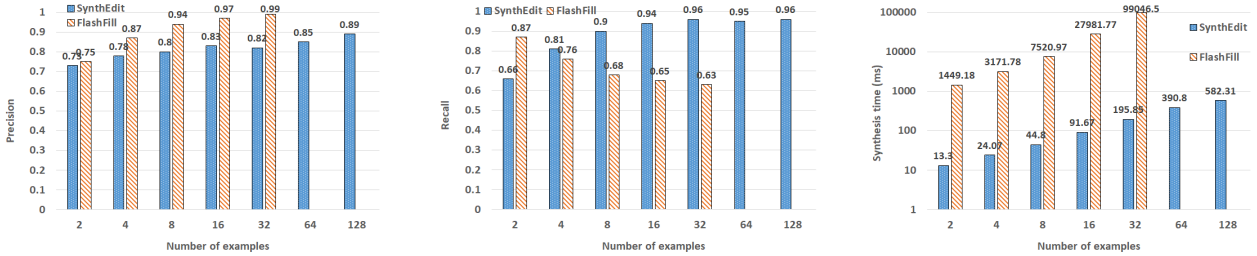
$SUB(\text{A}^{\text{s}}_{\emptyset}, Copy(\text{A}^{\text{s}}_{\emptyset}))$; $SUB(\text{W}^{\text{s}}_{\emptyset}, Copy(\text{W}^{\text{s}}_{\emptyset}))$; $SUB(\text{A}^{\text{s}}_{1}, Substr(\text{A}^{\text{s}}_{1}, 0, 1))$;
$INS(Const(\text{"."}))$; $SUB(\text{W}^{\text{s}}_{1}, Copy(\text{W}^{\text{s}}_{\emptyset}))$; $SUB(\text{A}^{\text{s}}_{2}, Copy(\text{A}^{\text{s}}_{2}))$;
$SUB(\text{W}^{\text{s}}_{2}, Copy(\text{W}^{\text{s}}_{\emptyset}))$; $SUB(\text{P}^{\text{s}}_{\emptyset}, Copy(\text{P}^{\text{s}}_{\emptyset}))$;
$SUB(\text{N}^{\text{s}}_{\emptyset}, Concat(Const(\text{"19"}), Copy(\text{N}^{\text{s}}_{\emptyset})))$; $SUB(\text{P}^{\text{s}}_{1}, Copy(\text{P}^{\text{s}}_{1}))$;
$SUB(\text{N}^{\text{s}}_{1}, Concat(Const(\text{"19"}), Copy(\text{N}^{\text{s}}_{1})))$; $SUB(\text{P}^{\text{s}}_{2}, Copy(\text{P}^{\text{s}}_{2}))$;

Learning from multiple examples: *SynthEdit* assumes the existence of several example instances from which it can learn multiple transformations. Although it is possible for a valid transformation to be synthesized from a small sample of the examples, it is not always possible to automatically identify the relevant sample and, therefore, all examples have to considered. Synthesizing transformations from multiple example instances that follow more than one format representation results in a *conditional transformation program*. To create such a program, we first partition the example instances into groups with source strings that follow the same format representation and synthesize a transformation for each partition[3]. Before transforming a new input string, we match its format representation against the ones for which we have synthesized transformations and apply the transformation of the matching format. If such a format representation does not exist the input string is left unchanged.

Complexity: There are two dominant tasks in *SynthEdit* in terms of complexity. Firstly, *EditSynthesis* runs in $O(m \times n)$ time [9], where m is the length of the source string and n is the length of the target string. Secondly, the generation of inverted index I implies the identification of the longest common sub-string between two token string values which is a dynamic programming problem that runs in $O(k \times l \times u \times v)$, where k is the number of source tokens, l is the number of target tokens, u is the source token value length, and v is the target token value length.

Synthesizing string expressions using `Expression Synthesis` only for the simplest token-based transformation, previously obtained using `EditSynthesis`, gives *SynthEdit* a computational

[3] If more than one transformation is possible per partition, we pick the one consistent with the majority of the example instances of that partition



(a) Average precision **(b) Average recall** **(c) Average synthesis time**

Figure 3: Experimental Results

advantage over algorithms such as *FlashFill* which considers all possible transformations that are consistent with the examples before ranking them and choosing the best one.

5 EVALUATION

In this section we perform a comparative evaluation[4] involving *SynthEdit* and an implementation of *FlashFill* from PROSE SDK[5], using 33 real world datasets[6], used in a related work [12]. Each dataset consists of up to 200 example instances from several domains such as person names, websites, songs, etc.

We report the average precision, recall and synthesis time over all datasets computed using k-fold cross-validation ($k = 10$) and various number of examples. At each iteration (fold), we synthesize a transformation program from n randomly picked example instances and test on the remaining instances. For the purposes of computing precision and recall, we count as a *true positive* any input string that is correctly transformed, i.e., the result of the transformation is similar to the expected output; as a *false positive* any input string that is incorrectly transformed; and as a *false negative* any input string that is left unchanged, i.e., there is no transformation synthesized for its format representation.

Comparative effectiveness: Avg. precision and recall as the number of examples varies. The precision results are shown in Figure 3a. *SynthEdit* achieves lower precision compared with *FlashFill* for the different numbers of examples. The difference can be explained by the ability of *FlashFill* to better generalize transformations as more examples are added by using a classifier trained on example instances. This allows it to correctly transform strings with format representations not covered by the examples. Conversely, *SynthEdit* performs a strict mapping between format representations and transformations and, therefore, requires at least one example instance for each format representation it transforms. For the last two cases, i.e., 64 and 128 examples, *FlashFill* required more RAM memory than was available.

The classifier employed by *FlashFill* to pick the right transformation given a new input string can become confused when some examples are too similar to each other in terms of the features used during learning. This means that for some input strings *FlashFill* fails to identify an appropriate transformation or the transformation picked is not consistent with the input, e.g., the transformation expects a type of token that is not present in the input. The consequence is a drop in recall visible in Figure 3b as the number of examples increases. By contrast, *SynthEdit* achieves better recall because more examples enables it to better differentiate between transformation cases.

Comparative efficiency: Avg. synthesis time as the number of examples varies. Figure 3c confirms the high complexity of *FlashFill* when the number of examples increases. Conversely, *SynthEdit* proves more than two orders of magnitude faster in synthesizing transformations. As opposed to *FlashFill*, *SynthEdit* does not aim to exhaust the search space of transformations for each example instance. Our algorithm uses edit distance computations to find the shortest path between the token-type representations of the source and target strings. Consequently, the transformation language is simpler but more efficient to learn.

6 CONCLUSIONS

We have contributed an effective and efficient solution to the problem of automating format transformation given input-output examples. We have used an edit distance based approach that identifies the shortest path from a source string to a target string and uses fuzzy matching of source and target tokens to generalize transformations applicable on new input strings, similar in format representation with the examples. Results from a comparative evaluation provide evidence that *SynthEdit* performs substantially more efficiently than the state-of-the-art while achieving better recall at the cost of slightly reduced precision.

Acknowledgement: This work is supported by the VADA Grant of the UK Engineering and Physical Sciences Research Council.

[4]Experiments were run on a 2.60 GHz Intel Core i7-4720HQ CPU with 8 GB RAM.
[5]https://microsoft.github.io/prose/
[6]www.microsoft.com/en-us/research/wp-content/uploads/2016/12/WebTableBenchmark.zip

REFERENCES

[1] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. Knowl. Data Eng.* 28, 5 (2016), 1217–1230.
[2] A. Bogatu, N. Paton, A. Fernandes, and M. Koehler. 2018. Towards Automatic Data Format Transformations: Data Wrangling at Scale. *Comput. J.* (2018).
[3] T. Furche, G. Gottlob, L. Libkin, G. Orsi, and N. W. Paton. 2016. Data Wrangling for Big Data: Challenges and Opportunities. In *EDBT*. 473–478.
[4] S. Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *POPL '11*. ACM, New York, NY, USA, 317–330.
[5] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. Narasayya, and S. Chaudhuri. 2018. Transform-data-by-example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (June 2018), 1165–1177.
[6] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *SIGMOD*. 683–698.
[7] M. Koehler, A. Bogatu, C. Civili, N. Konstantinou, E. Abel, A. A. A. Fernandes, J. A. Keane, L. Libkin, and N. W. Paton. 2017. Data context informed data wrangling. In *IEEE BigData, Boston, MA, USA, December 11-14, 2017*. 956–963.
[8] N. Konstantinou, M. Koehler, E. Abel, C. Civili, B. Neumayr, E. Sallinger, A. Fernandes, G. Gottlob, J. Keane, L. Libkin, and N. Paton. 2017. The VADA Architecture for Cost-Effective Data Wrangling. In *SIGMOD*. 1599–1602.
[9] M. Mohri. 2002. Edit-Distance of Weighted Automata. In *CIAA, 2002*. 1–23.
[10] M. Raza, S. Gulwani, and N. Milic-Frayling. 2014. Programming by Example Using Least General Generalizations. In *AAAI*. 283–290.
[11] R. Singh. 2016. BlinkFill: Semi-supervised Programming by Example for Syntactic String Transformations. *PVLDB* 9, 10 (June 2016), 816–827.
[12] E. Zhu, Y. He, and S. Chaudhuri. 2017. Auto-join: Joining Tables by Leveraging Transformations. *Proc. VLDB Endow.* 10, 10 (June 2017), 1034–1045.